VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

DOKUMENTACE KE SPOLEČNÉMU PROJEKTU DO IFJ A IAL

IMPLEMENTACE INTERPRETU IMPERATIVNÍHO JAZYKA IFJ16

Skupina 054, varianta a/3/II

11. prosince 2016

Rozšíření:

- **FUNEXP**

Rozdělení bodů:

Jan Věrný **(Vedoucí týmu) –** 20%	(xverny00)
Jan Oškera – 20%	(xosker02)
Antonín Vlach – 20%	(xvlach16)
Dominik Valachovič – 20%	(xvalac08)
David Zikmund – 20%	(xzikmu08)

Obsah

1. Úvod

Tato dokumentace popisuje implementaci interpretu imperativního jazyka IFJ16, který je podmnožinou jazyka JavaSE8. Implementaci interpretu byl zadán jako projekt do předmětů Formální jazyky a překladače a Algoritmy. Projekt byl rozdělen do čtyř hlavních bloků (lexikální analyzátor, syntaktický analyzátor, sémantický analyzátor, interpret), každému bude věnována samostatná kapitola. Dále balík obsahuje soubor ial.c, který obsahuje implementaci hashovací tabulky a třídicí funkci podle definice shell sortu.

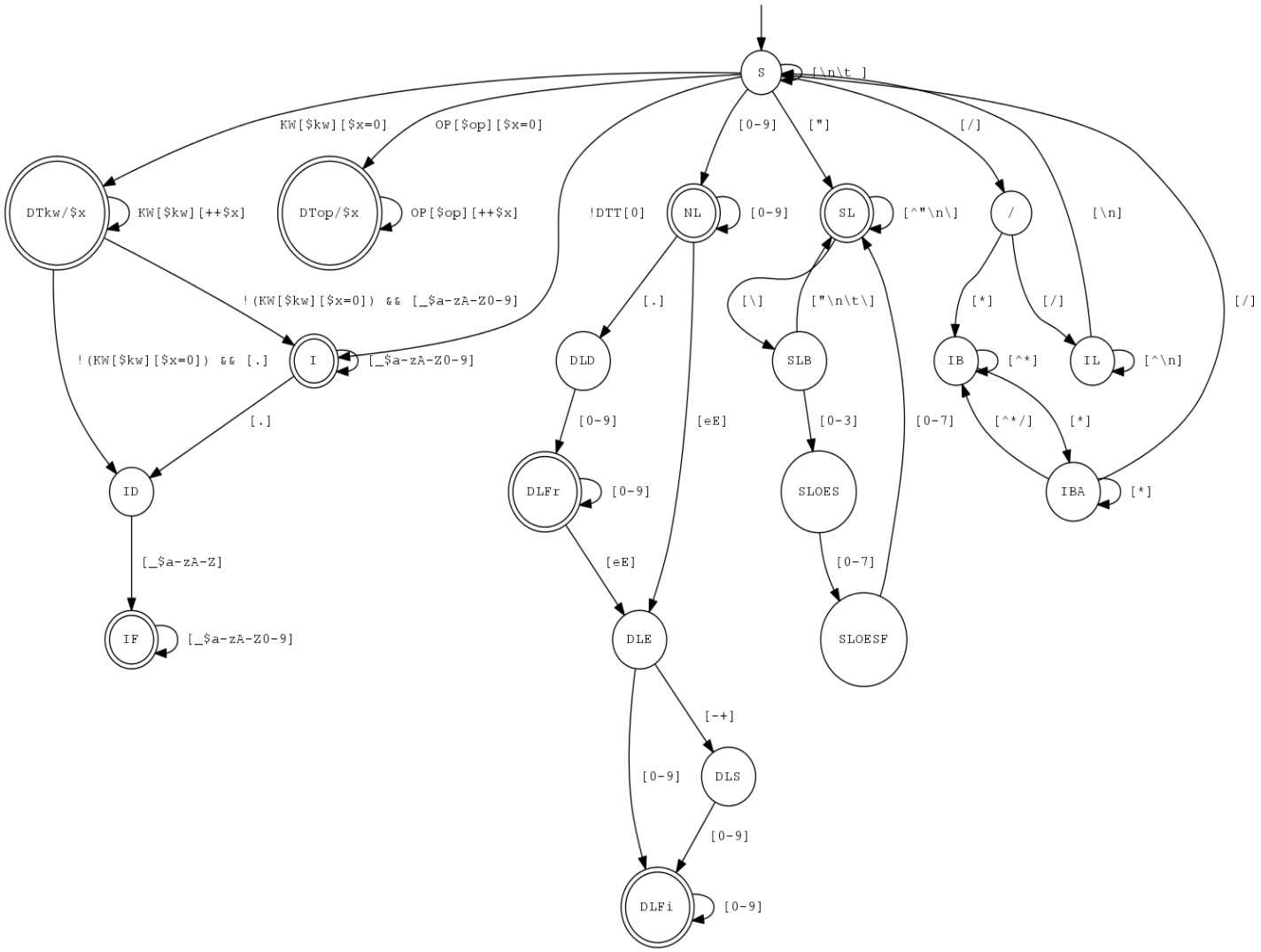
2. Struktura projektu

Projekt je rozdělen do čtyř bloků (lexikální analyzátor, syntaktický analyzátor, sémantický analyzátor, interpret). Každý blok byl vyvíjen nezávisle na těch ostatních. Naše řešení obsahuje soubor main.c. Ten zavolá syntaktickou analýzu (parser). Parser postupně volá lexikální analýzu, která zpracuje část zdrojového souboru a předá zpět token. Parser provede syntaktickou analýzu a uloží data do binárního stromu. Takto se projde celý zdrojový soubor. Nakonec parser předá strom sémantické analýze, která ho projde a zkontroluje ho. Když vše projde bez chyb, interpret provede kód. Některou alokovanou paměť si řeší hlavní bloky samy, zbytek řeší soubor mem_management. Soubor stack slouží jako pomocný k sémantické analýze (zásobník k procházení binárního stromu).

2.1. Lexikální analyzátor

Hlavním úkolem lexikálního analyzátoru je čtení zdrojového souboru a na základě pravidel lexikální analýzy, rozdělí soubor na lexémy, které následně předává syntaktické analýze. Zdrojový soubor je lexikální analýze předáván z hlavního souboru main. Lexémy jsou předávány spolu s načtenou hodnotou syntaktické analýze. Pokud se objeví lexikální chyba, vypíše se na standardní chybový výstup a syntaktickému analyzátoru se předá hodnota null (Podle zadání analýza pokračuje až dokonce, ikdyž už je přerušená chybou. Proto bylo nutné nepřerušovat a jen vracet hodnotu null.). Lexikální analyzátor je na principu konečného automatu (viz obrázek 1). Podle stavu automatu se pozná, jaký lexém jsme načetli.

Obrázek 1 – Lexikální analyzátor (konečný automat)



Legenda

- DTkw…datalessType(keyword)
- Dtop…datalessType(operator)
- /…divisionOperator(special state)
- DLD…doubleLiteralDot
- DLE…doubleLiteralExponent
- DLFi…doubleLiteralFinish
- DLFr…doubleLiteralFraction
- DLS…doubleLiteralSign
- I…identifier
- ID…identifierDot
- IF…identifierFinish
- IB…ignoreBlock
- IBA…ignoreBlockAsterisk
- IL…ignoreLine
- NL…numberLiteral
- S…start
- SL…stringLiteral
- SLB…stringLiteralBackslash
- SLOES…stringLiteralOctalEscapeSequence
- SLOESF…stringLiteralOctalEscapeSequenceFinish

- *KW[] = { "boolean", "break", "class", "continue", "do", "double", "else", "false", "for", "if", "int", "return", "String", "static", "true", "void", "while" };
- *OP[] = { "!=", "(", ")", "*", "+", ",", "-", ";", "<", "<=", "=", "==", ">", ">=", "{", "}"};

- $kw ∈ { 0, …, size(KW) }
- $op ∈ { 0, …, size(OP) }

2.2. Syntaktický analyzátor

Syntaktická analýza volá lexikální analyzátor a ten jí předá token (lexém s hodnotou). Poté na základě LL-gramatiky (syntaxe jazyka) a precedenční tabulky vyhodnotí, jestli je konstrukce jazyka správná. Vstupem syntaktické analýzy je binární strom, do kterého se ukládají zpracovaná data.

LL-gramatika

1: <file> -> <class-list>
2: <class-list> -> <class> <class-list>
3: <class-list> -> $
4: <class> -> class <id> { <class-item> }
5: <class-item> -> <function> <class-item>
6: <class-item> -> <staticvar> <class-item>
7: <class-item> -> $
8: <function> -> static <declaration> (<arg-list>) { <st-list> }
9: <arg-list> -> <arg> , <arg-list>
10: <arg-list> -> $
11: <arg> -> <declaration>
12: <st-list> -> <statement> <st-list>
13: <st-list> -> <localvar>
14: <st-list> -> $
15: <statement> -> <assignment> ;
16: <statement> -> <block>
17: <statement> -> <condition>
18: <statement> -> <cycle>
19: <statement> -> <call> ;
20: <statement> -> <return> ;
21: <assignment> -> <id> = <expression>
22: <block> -> { <block-list> }
23: <block-list> -> <statement> <block-list>
24: <block-list> -> $
25: <condition> -> if (<comparison>) <block> else <block>
26: <cycle> -> while (<comparison>) <block>
27: <call> -> <id>(<par-list>)
28: <par-list> -> <par> , <par-list>
29: <par-list> -> $
30: <par> -> <id>
31: <par> -> <expression>
32: <return> -> return <expression>
33: <parenthesis> -> (<expression>)
34: <expression> -> <parenthesis> + <expression>
35: <expression> -> <parenthesis> - <expression>
36: <expression> -> <term> + <expression>
37: <expression> -> <term> - <expression>
38: <expression> -> <term>
39: <term> -> <parenthesis> * <term>
40: <term> -> <parenthesis> / <term>
41: <term> -> <factor> * <term>
42: <term> -> <factor> / <term>

43: <term> -> <factor>
44: <factor> -> <call>
45: <factor> -> <literal>
46: <factor> -> <id>
47: <literal> -> <string>
48: <literal> -> <number>
49: <staticvar> -> static <declaration> ;
50: <staticvar> -> static <declaration> = <expression> ;
51: <localvar> -> <declaration> ;
52: <localvar> -> <declaration> = <expression> ;
53: <declaration> -> <type> <id>
54: <type> -> int
55: <type> -> string
56: <type> -> double
57: <id> -> class.id
58: <id> -> id
59: <comparison> -> <expression> == <expression>
60: <comparison> -> <expression> != <expression>
61: <comparison> -> <expression> < <expression>
62: <comparison> -> <expression> > <expression>
63: <comparison> -> <expression> <= <expression>
64: <comparison> -> <expression> >= <expression>

Obrázek 2 - Precedenční tabulka

	+	-	*	/	(	)	>	<	>=	<=	==	!=	id	$
+	>	>	<	<	<	>	>	>	>	>	>	>	>	>
-	>	>	<	<	<	>	>	>	>	>	>	>	>	>
*	>	>	>	>	<	>	>	>	>	>	>	>	>	>
/	>	>	>	>	<	>	>	>	>	>	>	>	>	>
(	<	<	<	<	<	>	E	E	E	E	E	E	<	E
)	>	>	>	>	E	>	>	>	>	>	>	>	E	>
>	<	<	<	<	<	E	>	>	>	>	>	>	<	>
<	<	<	<	<	<	E	>	>	>	>	>	>	<	>
>=	<	<	<	<	<	E	>	>	>	>	>	>	<	>
<=	<	<	<	<	<	E	>	>	>	>	>	>	<	>
==	<	<	<	<	<	E	>	>	>	>	>	>	<	>
!=	<	<	<	<	<	E	>	>	>	>	>	>	<	>
id	>	>	>	>	E	>	>	>	>	>	>	>	E	>
$	<	<	<	<	<	E	<	<	<	<	<	<	<	E

2.3. Sémantický analyzátor

Sémantický analyzátor je kontrola v interpretu, projde binární strom a na základě sémantiky vyhodnotí, jestli je vše v pořádku. Když ano, předá strom interpretu. Když ne vyhodí chybu na standardní chybový výstup.

2.4. Interpret

Vychytá poslední chyby (chyby interpretu) a pokud vše projde bez chyby, provede daný kód.

3. Řešení algoritmů z předmětu IAL

3.1. Shell sort

Shell sort využívá snižující se přírůstek. To znamená, že algoritmus neřadí prvky, které jsou přímo vedle sebe, ale prvky, mezi nimiž je určitá mezera, která je v každém kroku zmenšena. V okamžiku, kdy se velikost mezery sníží na 1, algoritmus se změní na insert sort a začne porovnávat sousední prvky. Výhodou tohoto poněkud komplikovaného přístupu je, že jsou prvky vysokých a nízkých hodnot velmi rychle přemístěny na odpovídající stranu pole.

4. Vývoj

Zde popíšeme, jak probíhal vývoj projektu a používané prostředky.

4.1. Rozdělení práce

Dominik Valachovič:	Lexikalni analyza
Jan Věrný:	Syntakticka analyza, interpret
Jan Oškera:	Semanticka analyza
Antonín Vlach:	ial.c, vestavene funkce, testovani
David Zikmund:	dokumentace + interpret

4.2. Použité prostředky

Pro komunikaci jsme používali Facebook, mobilní telefon a osobní setkání. Verzovali jsme přes webový verzovací systém GitHub.

5. Závěr

Projekt byl velmi časově náročný, hlavně z hlediska rozplánování jednotlivých částí. Naučilo nás to týmové spolupráci, a proto byl pro naši skupinu velkým přínosem.